Holons: towards a systematic approach to composing systems of systems

Gordon Blair, Yérom-David Bromberg, Geoff Coulson, Yehia Elkhatib, Laurent Réveillère, Heverson Borba Ribeiro, Etienne Rivière, François Taïani

Holons: towards a systematic approach to composing systems of systems

Gordon Blair[1], Yérom-David Bromberg[2], Geoff Coulson[1], Yehia Elkhatib[1],
Laurent Réveillère[3], Heverson B. Ribeiro[4], Etienne Rivière[4] and François Taïani[2]
1. University of Lancaster, UK — 2. University of Rennes 1 - IRISA, France
3. University of Bordeaux, France — 4. University of Neuchâtel, Switzerland
g.coulson@lancaster.ac.uk and etienne.riviere@unine.ch

ABSTRACT

The world's computing infrastructure is increasingly differentiating into self-contained distributed systems with various purposes and capabilities (e.g. IoT installations, clouds, VANETs, WSNs, CDNs, ...). Furthermore, such systems are increasingly being *composed* to generate *systems of systems* that offer value-added functionality. Today, however, system of systems composition is typically ad-hoc and fragile. It requires developers to possess an intimate knowledge of system internals and low-level interactions between their components. In this paper, we outline a vision and set up a research agenda towards the *generalised programmatic construction* of distributed systems as compositions of other distributed systems. Our vision, in which we refer uniformly to systems and to compositions of systems as *holons*, employs code generation techniques and uses common abstractions, operations and mechanisms at all system levels to support uniform system of systems composition. We believe our holon approach could facilitate a step change in the convenience and correctness with which systems of systems can be built, and open unprecedented opportunities for the emergence of new and previously-unenvisaged distributed system deployments, analogous perhaps to the impact the mashup culture has had on the way we now build web applications.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; D.2.10 [**Software Engineering**]: Design—*Methodologies*

General Terms

Design, Algorithms, Standardisation

Keywords

System composition, Systems of systems, Distributed systems

1. INTRODUCTION

The world's computing infrastructure has become far removed from the traditional picture of PCs, mobile devices and IP networks; it now subsumes a diverse range of (semi-) autonomous infrastructures and sub-systems with widely varying capabilities. Furthermore, it is increasingly common for these systems to *interact*. For example, VANETs talk to IoT environments that underpin smart cities [7], overlays adapt when the underlying network changes [9], and cloud infrastructures store and process data gathered from WSNs [16]. Other prominent examples involve the so-called "cloud of things" [12, 26], and the "TerraSwarm" [5]: the future mega-environment of trillions of interacting sensors and actuators.

However, when we consider techniques available for the construction of such *systems of systems* [20], we see a significant deficiency in the state of the art. In particular, when developing individual distributed systems, developers are forced to focus strongly at the level of individual nodes (e.g. designing the per-node behaviour of a new DHT); so when they develop a system of systems they are similarly forced to focus on the internals of the systems being composed (e.g. how individual nodes of the two systems must interact). But as the number and diversity of distributed system deployments burgeon, such low-level practices are increasingly miring developers in accidental complexities, and hampering the development of new systems of systems. We believe we must stop reasoning in terms of individual nodes, or even individual systems, and move away from ad-hoc approaches to system of systems composition.

We propose in this paper a vision towards the construction of distributed systems of systems that uniformly addresses the specification of *both* individual distributed systems *and* their composition, using a common code generational approach. In a nutshell, our approach is as follows:

1. To represent systems as *first-class* entities that we can specify and handle as programmatic units – i.e. at a level that hides their internals, and in particular the individual behaviour of their constituent nodes;
2. To treat the composition of such representations as *a simple operation* that yields a new unitary first-class system.

This is clearly an ambitious goal. We postulate, however, that distributed systems research suggests the feasibility of our proposed approach. In particular, we draw inspiration from: i) component frameworks; ii) generative programming; and iii) gossip-based self-stabilising overlays.

We believe our vision has the potential for cardinal impact on the way future systems of systems are constructed, and see it opening unprecedented opportunities for previously-unenvisaged distributed system deployments – perhaps analogous to the impact that the mashup culture has had on

the way we now build web applications. We hope that the vision, challenges, and tentative solutions we outline in this paper might serve as a roadmap to help structure future community-wide research in the field.

In the remainder of the paper, Section 2 discusses related work; Section 3 presents some motivational use cases; and Section 4 outlines a proposed model of first-class composable systems. Finally, Section 5 concludes, and discusses future work and open problems.

2. RELATED WORK

We consider relevant related work under the headings of component frameworks, generative programming, and gossip-based self-stabilising overlays.

Component frameworks. Component-based software models (e.g. DCOM, EJB, CCM, RAPIDWare [23], OpenCom [4], and Fractal [3]) have long offered a popular approach to the construction of distributed systems by composition, and we see such work as having paved the way for compositional principles at the systems of systems level. However, such efforts have focused on composition at the level of nodes or sub-node software units, rather than at the global level of whole distributed systems. Similarly, while interesting efforts were produced towards the automation of service composition [17] the target remains that of individual nodes rather than systems as a whole. Our contention is that this is not the appropriate level of abstraction to apply when considering the construction of systems of systems.

Generative programming. Recognising the need to work at a higher level of abstraction, several researchers have investigated the definition of individual systems through high-level specification and code generation. This is especially the case in the fields of network overlays (e.g. P2 [19], Mace [15], Mosaic [22], SLOSL [1]); and wireless sensor networks (e.g. Kairos [11], Flask [21]). Such work employs a domain-specific programming model that abstracts some of the detail and complexity of inter-node communication and coordination, the goal being to write a single specification from which per-node code can be generated and deployed. For example, Flask uses a variant of Haskell, P2 uses a variant of Datalog, and SLOSL uses an SQL-based notation. While the generative programming approach is an advance over hand-coded, node-level system construction, it still tends to focus on the specification of per-node behaviour (e.g. reacting to incoming messages from other nodes, or to lost links or membership changes). Unfortunately, this limits its applicability to system of systems composition: it is still concerned with internals, not with externally-facing perspectives, and so still forces the developer to focus on the internals of the systems being composed rather than freeing her to think at the level of systems as wholes.

Some overlay-based generative programming designs go a little further towards facilitating systems of systems composition. For example, Dynamis [25] and SpiderNet [10] seek to build compositions using a distributed probing approach that "opportunistically" seeks nodes from other overlay instances, attempting to find high-quality service paths through which instances can be linked. The key limitation of these designs is a lack of *generality*. Firstly, they support (opportunistic) composition at run-time, but not at system specification time. Secondly, although they support "horizontal" composition (i.e. bridging) they lack support for "vertical" composition (i.e. layering one system over another). More fundamentally, they lack support for composing different *classes* of systems (e.g. composing a WSN with a cloud).

Gossip-based self-stabilising overlays. Work in this third area is of interest because it offers an uniform means of constructing a diverse range of distributed systems, e.g. robust routing [8], publish/subscribe [28] and generic structured overlays [13, 14, 29]: this promises a means of constructing distributed systems that is more conducive to system composition than other approaches [24]. Such work especially focuses on the construction of vertically-composed complex systems such as rings whose nodes are implemented internally as stars or sub-rings, or compositions of broadcast trees over different IP domains [27]. However, it does not offer much in terms of other types of composition discussed above (i.e. "opportunistic", "horizontal"): the behaviours and topologies supported by these systems are still primarily monolithic and isolated. Furthermore, the focus is at the level of mechanisms as opposed to programmatic construction.

3. MOTIVATING USE CASES

To further motivate and exemplify system of systems composition, we present in this section two typical use cases, followed by a brief discussion. **Irrigation** is an agricultural monitoring and actuation scenario; and **Rescue** deals with the opportunistic composition of potentially-isolated rescue teams in a mountain rescue scenario.

3.1 Irrigation: WSN-enhanced agriculture

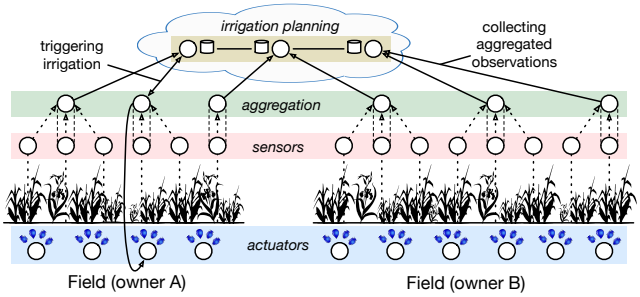

Figure 1: Irrigation use case.

Our first use case (see Figure 1) is an agricultural scenario in which battery-operated *sensors* are distributed over fields to collect data such as humidity, crop height or soil chemistry. Each sensor is equipped with short-range communications technology with which it can communicate with its peers in the vicinity. Some sensors are additionally equipped with a long-range cellular communication capability, and so can serve as *aggregators* that collect data locally and ship it to a *planning system* running in a remote cloud. The "loop" is closed when the planning system drives an in-field *actuator network* that controls irrigation valves, fertiliser release, etc. As well as being driven by the planning system, the actuators may also be more locally driven by computations carried on the actuator nodes themselves and/or nearby sensor nodes with spare computational capacity. This enables the long-range links to be used sparingly to conserve power where necessary.

Commentary: We have here a technology-rich environment with numerous connectivity options and possibilities for adapted behaviour depending on resource availability (e.g. "switch to local planning when aggregator power is running low"). It is easy to see how factoring out the various areas of distributed functionality into composable systems (e.g. a

tree-based sensor network, a mesh-based aggregation network, a system of local planning modules running on a subset of sensor/actuator nodes, etc.), and then composing these as unitary entities, might considerably ease the development of practical deployments compared to the complexity evident at the node level.

This use case illustrates two of the flavours of system composition that we discussed in Section 2: *vertical* (e.g. layering of the aggregation network over the sensor network) and *horizontal* (e.g. Field A to the planning system to Field B). It does not seem to require opportunistic system composition, but it does have a strong requirement for resource-driven adaptation and also probably has a requirement to *evolve* over time – e.g. depending on the season, whether more or fewer fields are in use, etc.

3.2 Rescue: Connecting teams using a FANET

Our second use case (see Figure 2) involves a natural disaster setting in which rescue teams are deployed over a large area such as a mountain. If we assume that the rescuers need to use a coordination and information sharing application using mobile ad hoc network (MANET)-based communication, and that they may need to range widely, we can see that individuals and teams could easily become isolated, both from each other and from backhaul connectivity to the remote Control centre. To alleviate this, we might deploy a swarm of Micro Air Vehicles (MAVs) over the area, and have them self-organise into a Flying Ad-Hoc Network (FANET [2]). If the MAVs are equipped with long-range 3G/satellite capability, they can not only horizontally compose (i.e. bridge) isolated teams, but also enable the teams to communicate with the Control centre.

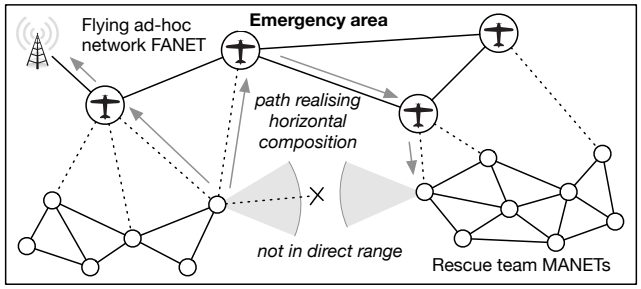

Figure 2: Rescue use case.

Commentary: Developing the software for this scenario involves system of systems composition. It also requires *opportunistic* composition: the rescue team systems must be "on the lookout" for the appearance of other systems such as the FANET, and be ready to compose with it when it appears. Furthermore, such composition should be engineered in such a way as to minimise bottlenecks and single-points-of-failure (e.g. by maintaining redundant links). Expressing opportunistic and redundant-link functionality would be a complex task indeed if addressed at the level of individual nodes. It is clear that attacking the problem at the level of system composition, aided by code generation for the node level, has considerable potential.

3.3 Discussion

We summarise the various properties of our use cases in Table 1. An initial observation concerns *heterogeneity*: system of systems programmers must typically deal with a range of device types, each with specific features. It is therefore *ab-initio* clear that working at the device level will be a time-consuming and error-prone approach. Next, it is easy to extrapolate from the use cases that both *vertical* and *horizontal* composition are likely to be common, very possibly co-existing as illustrated by Irrigation. Moreover, Rescue shows that a single system type may be instantiated multiple times, further increasing composition complexity.

		Irrigation	Rescue
Heterogeneity		✓	✓
Composition	*vertical*	✓	
	horizontal	✓	✓
	opportunistic		✓
Adaptivity		✓	✓
Evolution		✓	

Table 1: Challenges highlighted by the use cases.

We have also seen the importance of *opportunistic* composition whereby systems perceive at run-time that they may benefit from composing with other systems, either vertically, horizontally or both. In Rescue, the "trigger" for opportunistic composition was the detection of the physical proximity of the FANET and the discovery that it provides similar connectivity functionality to the set of MANETs, but with extended coverage area. We can easily conceive of generalising such triggers to what has been called "semantic proximity" [5]. For example, in Irrigation, we may want to compose two actuator systems in adjacent fields when the planning system deems that conditions in the two fields are sufficiently similar.

Finally, we observe that many systems of systems will be *adaptive* or *evolutionary* or both. We use the term "adaptive" to refer to systems that need to reconfigure themselves autonomously and dynamically, as in the case of the FANET system in Rescue. We use the term "evolutionary" to refer to situations where the need arises for users to explicitly alter the set of deployed systems and their compositions over time, as seen especially in Irrigation.

4. PROPOSED APPROACH

In order to address the above challenges, we now present a "straw-man" design intended as a first attempt at a principled and systematic approach to the programmatic composition of systems of systems. Our first principle is to model any distributed system as a unitary first-class programmatic entity that we call a *holon*[1], that can be specified, manipulated, and reasoned about in a program; and then to provide programmatic concepts that enable a developer to construct new holons – i.e. systems of systems – through programmatic holon composition. This composition process is intended to be very simple and straightforward, requiring only a few program lines or simple graphical tools.

An important aspect of our approach is that the developer is empowered to reason about system of systems definition at the level of whole systems (holons), avoiding any need to explicitly manipulate the node-level code that underlies

[1]We adopt this term from Arthur Koestler's book, "The Ghost in the Machine" (1967), where it is used to refer to a member of a hierarchy that is a whole when viewed from below, and simultaneously a part when viewed from above. The term has been co-opted for use in the computer science field before [6], but in a manner unrelated to our use.

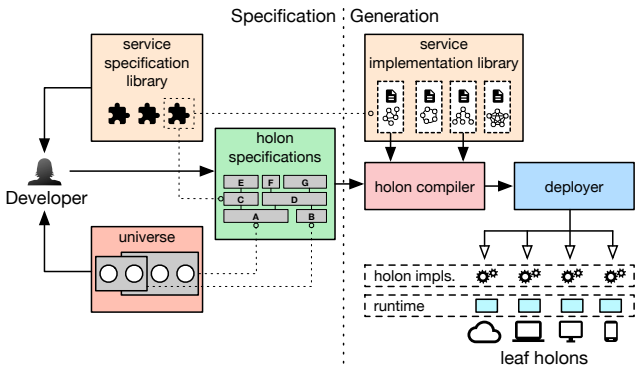

Figure 3: General architecture of the proposed approach.

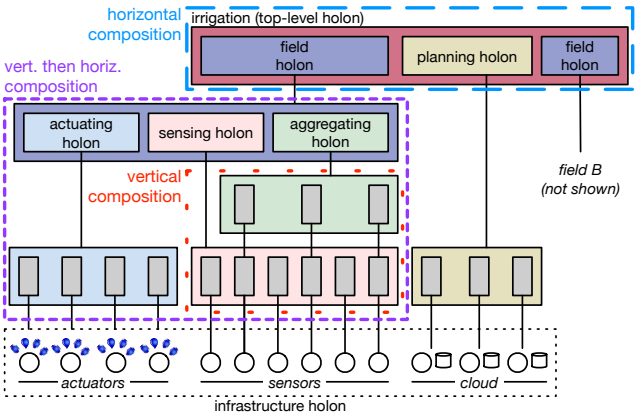

Figure 4: Vertical and horizontal composition in Irrigation.

the holons being composed[2]. This avoids the "abstraction mismatch" that we identified in Section 2.

We detail in the remainder of this section the key constituents of our approach, which are depicted in Figure 3.

4.1 Specifying holons

As outlined, a distributed system, as represented as a holon, is a recursive, hierarchical, composition of other systems (holons). The holons at the level immediately below a given holon are referred to as the latter's *sub-holons*. The hierarchy bottoms out at the level of the smallest possible "systems" (holons) in our model, which are degenerate distributed systems that run on individual physical nodes; these are known as *leaf holons*.

In our model, the *vertical composition* of holons is achieved by specialising or piling up holons on top of one another. For example, Figure 4 illustrates an *aggregating* holon that is built on top of (vertically composed with) a *sensing* holon. Is is the fact that both of these holons are stacked on top of a common underlying set of leaf holons (i.e. sensors) that defines this as a vertical composition. *Horizontal composition*, on the other hand, occurs when the sub-holons of a newly-defined holon are built over disjoint sets of underlying leaf holons. This can be seen in the *irrigation* holon, which horizontally composes *field* holons and the *planning* holon, each of which builds on disjoint sets of leaf holons.

4.1.1 Specifying a holon's sub-holons

A holon's constituent sub-holons are selected dynamically from among a set of candidate sub-holons on the basis of dynamically-valued *properties* attached to the latter (we defer a detailed discussion of properties to Section 4.1.2). Candidate sub-holons are selected from a so-called *base holon* which serves as a kind of "platform" on which the new holon is specified. Any holon can be used as a base-holon, but we define two "special" holons that may serve as suitable base holons in many cases: i) the *infrastructure holon*, whose sub-holons are all the leaf holons ever defined – this essentially represents a dynamic catalogue of available primitive system elements; and ii) the *universe holon*, whose sub-holons are *all* holons ever defined (except the universe holon itself), including the infrastructure holon, all leaf holons, and all already-defined and future-defined holons.

The fact that the set of sub-holons that will comprise a newly-defined holon is selected intentionally (i.e. according to dynamic selection criteria) is key. It means, for example, that the compositional process is not limited to selecting sub-holons that are newly-instantiated and built from scratch: it is also possible to select sub-holons that are *already deployed and running*. Furthermore, the sub-holons selected are not necessarily known at holon specification time: it is also possible to employ *opportunistic* selection at run-time, as conditioned by run-time considerations (expressed as properties) such as QoS requirements, functionalities, available energy, reachability to other holons, etc.

4.1.2 Specifying a holon's service

We have so far discussed the abstract composition of holons, but have said nothing about the specific functionality that a deployed holon will offer – i.e. what the holon will actually *do*. We refer to this functionality as the holon's *service* – essentially, it defines the value-added functionality that the new holon will provide on top of the services already offered by its sub-holons.

In the general case, the developer chooses her new holon's service from a *service specification library*, which contains reusable predefined service specifications. Service specifications are *abstract*: the actual implementation of a service is transparent to the developer, allowing her to think globally about her holon's functionality rather than at the level of its constituents. Furthermore, dealing in terms of abstract services allows us to offer several alternative implementations of a given service (these implementations are kept in a *service implementation library*); in such cases it can be left to the compiler to choose the most appropriate implementation depending, e.g., on QoS considerations or potential for reusing existing services already deployed on the target sub-holons.

An abstract service specification is annotated with "required" properties, called *rprops*, which represent functional and non-functional properties that the holon's service will require from its sub-holons to provide a certain level of service or functionality. Rprops come in two flavours: *pre-deploy* rprops should be satisfied prior to deployment (i.e. at compile/link time) and remain satisfied subsequently at run-time; whereas *post-deploy* rprops need not be satisfied at compile/link time, but the deployment/ runtime system (see Section 4.3) should make best efforts to satisfy them opportunistically at run-time. In addition, service specifications are annotated with *property dependency rules* that detail what it takes for each rprop to be satisfied in terms of functionality offered by the holon's sub-holons. These

[2]Of course, this code still has to be produced, but it is encapsulated in libraries (see more detail below) and hidden from the developer who is specifying system compositions.

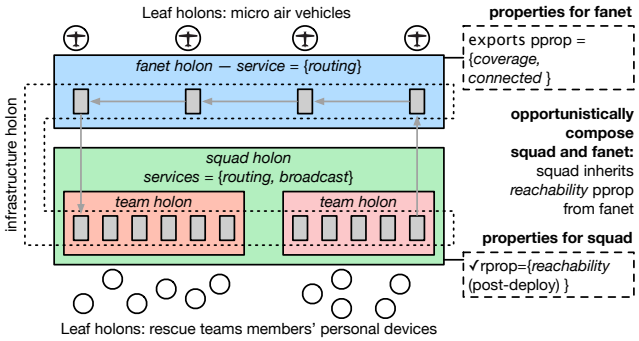

Figure 5: Provision of a pprop by opportunistic composition.

property dependency rules are written in terms of "provided" properties called *pprops* that are attached to the (assumed to be already-existent) sub-holons. In turn, the holon's *own* pprops are asserted when all these property dependency rules are satisfied.

4.1.3 Specification example

We provide an example of service specification based on the Rescue use case; see Figure 5. Specifically, we focus on the case of the *squad* holon bridging multiple *team* holons.

Let us first assume that the developer has selected a service called *routing* for both the *team* and *squad* holons: this offers a point to point routing service between sub-holons. The *routing* service specification defines an rprop called *reachability* that captures a requirement that all of the sub-holons must always be mutually "reachable". It also defines a corresponding property dependency rule that grounds *reachability* by stating that all of the sub-holons must offer either a *connected* or a *coverage* pprop.

Let us now address the scenario in which reachability across widely-distributed teams might not necessarily be provided on initial deployment. To accommodate this scenario, we consider *reachability* as a pre-deploy pprop for the *team* holons, but as a post-deploy pprop for the *squad* holon. Given this, when the subsequent deployment of the FANET makes available a *fanet* holon that provides alternative reachability across teams via a *coverage* pprop, the *squad* holon's service's property dependency rule can trigger the opportunistic runtime composition of the *squad* and *fanet* holons, enabling the former's *reachability* property to be satisfied by the latter.

4.2 Compiling holons

Once specified, holons are compiled into code that will run on physical nodes. The compiler builds a per-node executable relating to the holon by composing code modules taken from the service implementation library, according to the system's holon hierarchy. This composition must respect any property dependency rules, ensuring that rprops (e.g. *reachability*) can be safely underpinned by a suitable combination of sub-holon pprops (e.g. *connected* or *coverage*), and the associated holon's own value-adding code. From a bottom-up perspective, this process begins with leaf holon pprops such as *stability*, *access to a persistent source of energy*, etc.

The compiler should only generate code for a given holon if it is assured (under the system's working assumptions) that all pre-deploy rprops in the holon's entire underlying hierarchy can be satisfied. On the other hand, the compiler is allowed to ignore post-deploy rprops that it cannot enforce at compile time, and to defer their (possible) satisfaction until run time, at which time they might be satisfied by compositions with other running holons discovered opportunistically by the runtime, as described above.

We see considerable scope for automated optimisation in the composition of holons at both compile time and run time. The compiler could for instance exploit property dependency rules to infer situations in which requirements on sub-holons might be met in indirect ways. Furthermore, as we assume that most service implementations will employ overlay network structures based on gossip-based self-stabilising overlays, the compiler can potentially extract similarities between holons' network structures and merge them into common structures for better robustness and/or lower costs [18].

4.3 Deployment and runtime support

At runtime, holons require coordination and communication mechanisms that can support the different types of composition (vertical, horizontal, opportunistic) that we have discussed earlier. This support should further be scalable, efficient and robust to sustain the large-scale deployment scenarios we have mentioned (smart cities, e-agriculture, large-scale rescue operations). We plan to fulfil these needs by exploiting a combination of self-organising overlays [13,28], epidemic protocols [27], and well-chosen point-to-point and multicast interactions, dynamically selected depending on the scope and scale of required interactions.

In architectural terms, such mechanisms will be managed in a *distributed runtime*, deployed on each physical node. The key responsibilities of the runtime are as follows:

- *Metadata management*: The runtime will keep track of which holons exist on which nodes, along with per-node dynamic properties (hardware capabilities, power status, configuration etc.) and holons (service type, pprops, etc.).
- *Deployment service*: Whenever a request is made to instantiate a holon on new hardware nodes, the runtime will populate the nodes with the appropriate leaf holon code and start the appropriate services.
- *Opportunistic composition*: The runtime will implement a *discovery service* that actively seeks for non-local holons in order to perform opportunistic compositions – i.e. compositions that derive from post-deploy rprops.

5. CONCLUSIONS AND FUTURE WORK

This paper proposes a new paradigm for the construction of distributed systems of systems. Our vision hinges on two key ideas: (i) treating any kind of distributed system as a first class programmatic entity (which we have termed a *holon*); and (ii) using sub-holon selection and composition as fundamental operations on holons to generate further holons, both at compile time and run time. These ideas permeate our proposed approach to the specification, deployment, and management of distributed systems of systems in a principled and unitary fashion. We believe that our approach has strong potential to raise the level of abstraction of distributed programming by focusing explicitly on *systems of systems* rather than merely on systems (or, worse, merely on nodes).

Besides its abstracting power, one of the key elements of our vision is its unifying nature: as it provides a uniform architectural view of diverse types of systems, it can be applied equally well to application-level software and to low-level infrastructure. As well as simplifying the task of the system of systems developer, this architectural uniformity naturally exposes opportunities for optimisation (e.g. the sharing of

common communicational structures and services among holons), assisted by the runtime's support for opportunistic composition and property-based reasoning.

We believe our vision has potential as a convergence point for future system of systems research. In particular, we close by suggesting the following set of open issues within which research groups with diverse interests and expertise can help bring the holon vision to fruition:

– *Domain specific languages and associated design/ development tools.* We have outlined a schema for holon specification, but have said little about notations/ abstractions/ processes to help programmers generate these specifications. We envision a range of DSLs that might be suited to different applications and domains.
– *Type systems to control composition.* Our uniform approach to system specification lends itself to formal type-checking: conformance rules could be associated with the sub-holon selection and composition processes to enforce sound compositions, and to formalise dependencies. We consider that type verification could have a big impact on developers' ability to manipulate and reason about increasingly complex distributed systems.
– *Security and privacy* will obviously be critical concerns in a world that makes it "easy" to compose multi-levelled distributed systems. Although security and privacy are highly challenging issues, we think the artefacts of our design may help in the provision of hooks for the implementation of security and privacy policies – e.g. a protected runtime might be trusted to generate only certified compositions.
– *Service composition.* Our approach is underpinned by a library of service specifications that can be composed in many configurations and still work correctly. Achieving this is not straightforward. We are clear that gossip-based protocols offer a promising basis for this, but significant work remains in this area.

6. REFERENCES

[1] S. Behnel. SLOSL: A modelling language for topologies and routing in overlay networks. PDP, 2007.

[2] I. Bekmezci, O. K. Sahingoz, and S. Temel. Flying ad-hoc networks (FANETs): A survey. *Ad Hoc Networks*, 11(3), 2013.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. Stefani. The FRACTAL component model and its support in java. *Soft.: Pract. and Exp.*, 36(11), 2006.

[4] G. Coulson, G. Blair, P. Grace, F. Taïani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Transactions on Computer Systems*, 26(1), 2008.

[5] E. Lee *et al.* The swarm at the edge of the cloud. *IEEE Design & Test*, 31(3), 2014.

[6] K. Fischer, M. Schillo, and J. Siekmann. Holonic multiagent systems: A foundation for the organisation of multiagent systems. HoloMAS, 2003.

[7] P. Ghazizadeh, R. Mukkamala, and S. El-Tawab. Scheduling in vehicular cloud using mixed integer linear programming. MSCC, 2014.

[8] C. Gottron, S. Bergsträsser, and R. Steinmetz. Robust overlay routing in structured, location aware mobile peer-to-peer systems. MOBIQUITOUS, 2013.

[9] P. Grace, D. Hughes, B. Porter, G. S. Blair, G. Coulson, and F. Taïani. Experiences with Open Overlays: A middleware approach to network heterogeneity. EuroSys, 2008.

[10] X. Gu, K. Nahrstedt, and B. Yu. SpiderNet: An integrated peer-to-peer service composition framework. HPDC, 2004.

[11] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. DCOSS, 2005.

[12] C. Huo, T.-C. Chien, and P. Chou. Middleware for IoT-cloud integration across application domains. *IEEE Design & Test*, 31(3), 2014.

[13] M. Jelasity, A. Montresor, and O. Babaoglu. T-Man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13), 2009.

[14] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM TOCS*, 25(3), 2007.

[15] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. MACE: Language support for building distributed systems. PLDI, 2007.

[16] K. Lee, D. Murray, D. Hughes, and W. Joosen. Extending sensor networks into the cloud using Amazon Web Services. NESEA, 2010.

[17] L. Leite, C. Moreira, D. Cordeiro, M. Gerosa, and F. Kon. Deploying large-scale service compositions on the cloud with the choreos enactment engine. NCA, 2014.

[18] S. Lin, F. Taïani, and G. Blair. Exploiting synergies between coexisting overlays. DAIS, 2009.

[19] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. SOSP, 2005.

[20] M. W. Maier. Architecting principles for system of systems. *Systems Engineering*, 1(4), 1998.

[21] G. Mainland, G. Morrisett, and M. Welsh. Flask: Staged functional programming for sensor networks. ICFP, 2008.

[22] Y. Mao, B. T. Loo, Z. Ives, and J. M. Smith. Mosaic: Declarative platform for dynamic overlay composition. *Computer Networks*, 56(1), 2012.

[23] P. K. Mckinley, U. I. Padmanabhan, N. Ancha, and A. Ancha. Experiments in composing proxy audio services for mobile users. Middleware, 2001.

[24] E. Rivière, R. Baldoni, H. Li, and J. Pereira. Compositional gossip: A conceptual architecture for designing gossip-based applications. *OSR*, 2007.

[25] F. A. Samimi and P. K. McKinley. Dynamis: Dynamic overlay service composition for distributed stream processing. SEKE, 2008.

[26] J. Soldatos, M. Serrano, and M. Hauswirth. Convergence of utility computing with the internet-of-things. IMIS, 2012.

[27] F. Taïani, S. Lin, and G. S. Blair. Gossipkit: A unified component framework for gossip. *IEEE Transactions on Software Engineering*, 40(2), 2014.

[28] S. Voulgaris, E. Rivière, A.-M. Kermarrec, and M. van Steen. Sub-2-Sub: Self-organizing content-based publish subscribe for dynamic large scale collaborative networks. IPTPS, 2006.

[29] S. Voulgaris and M. van Steen. VICINITY: A pinch of randomness brings out the structure. Middleware, 2013.